

Getting started with ASDF

by Mario S. Mommer*

June 1, 2004

1 Introduction

The aim of this document is to give a brief introduction to the use of **ASDF**, **A**nother **S**ystem **D**efinition **F**acility. It is not about the arcane tricks and tips, and is not about the design of **ASDF** itself nor about system definition tools in general.

Defining a system is in the vast majority of cases usually trivial, so hacking up some minimal script will usually work. This is why many never take the time to learn a system definition facility. To acknowledge this, we concentrate on the simple cases, and on the general usage, hoping to yield a high return for a minimal investment. We leave the more advanced features to the documentation of **ASDF** [1].

1.1 What problem does **ASDF** solve?

When you download the source code of some software from the internet, or get it from some other source, you usually do not get an amorphous bunch of files, but instead get a system of components that depend on each other in some particular way. The consequence of this is that, if you want to build the software (be it a library, or be it an application), you will probably have to build these components, and the components of these components in order, perhaps giving some special treatment to some of them. You would, of course, be very grateful if the developer had prepared everything, and you could trigger the build process by a single command.

If you are a developer working in a project with a few components, you will probably want some mechanism that keeps track of the dependencies between these components, so that if you change one component, triggering a rebuild only recompiles and reloads the components which are affected.

Finally, you probably want a consistent way of dealing with the dependencies between components and of building and loading software systems, simply because it saves everyone time when installing software and using software.

*Email: mommer@igpm.rwth-aachen.de

ASDF is, roughly speaking, an extensible facility for defining the dependencies between software components, and specifying eventual details of the build process. It is also in fairly wide use, so that you can assume that your system definition will be understood by many others.

The same can be said about `mk:defsystem`, which has fans as well as detractors. We will only concentrate here on ASDF, since we have to start somewhere. The difference between these two only become aparent for the power user.

2 Defining systems with ASDF

An ASDF system definition is stored in a file with the extension `.asd`. To make the discussion clearer, we are going to pretend that we want to write the system definition facility for our software project called, unassumingly, `cow`.

The system definition file should be called `cow.asd`. If you use emacs, you might want to put the following as the first line in that file

```
;;; -*- Mode: Lisp; Syntax: ANSI-Common-Lisp; Base: 10 -*-
```

which makes sure that the proper syntax support is turned on.

After that, `cow.asd` should start with the following code (preceded or followed perhaps by comments; goes almost without saying)

```
(defpackage #:cow-asd
  (:use :cl :asdf))
```

```
(in-package :cow-asd)
```

The next thing is to write a `defsystem` form, together with some (optional) extra information.

```
(defsystem cow
  :name "cow"
  :version "0.0.0"
  :maintainer "T. God"
  :author "Desmon Table"
  :licence "BSD sans advertising clause (see file COPYING for details)"
  :description "Cow"
  :long-description "Lisp implementation of our favorite ruminant")
```

As you might have guessed, only the first line is mandatory.

2.1 the base case

In the simplest case you have a directory with a few files. Suppose that in our project, we have the following scenario. We have the file `legs.lisp`, `tail.lisp`, and `head.lisp`. Then there is this file called `package.lisp`, from which, oddly,

everything else depends. Suppose that also `tail.lisp` depended on `legs.lisp`, for some mysterious reason. Our `defsystem` form would look like this.

```
(defsystem cow
  ;;; (Optional items omitted)
  :components ((:file "tail"
                 :depends-on ("package" "legs"))
               (:file "legs"
                 :depends-on ("package"))
               (:file "head"
                 :depends-on ("package"))
               (:file "package")))
```

That's all.

In this base case, you would keep the file `cow.asd` in the same directory as the source files. If you want to try out our nice `cow` system, you can jump directly to section 3.

2.2 A system with modules

Suppose that we have the following, more involved structure. We have a `head.lisp`, and a `legs.lisp`. But we also have a subsystem called `respiratory`, which consists of a few more files, and lives in its own subdirectory called `breathing`. Similarly, we have another subsystem called `circulation`. The `defsystem` form could look more or less like this.

```
(defsystem cow
  :components ((:file "head" :depends-on ("package"))
               (:file "tail" :depends-on ("package" "circulation"))
               (:file "package")
               (:module "circulation"
                 :components ((:file "water"
                                   :depends-on
                                   "package")
                               (:file "assorted-solids"
                                   :depends-on
                                   "package")
                               (:file "package"))
               (:module "breathing"
                 :components (...))))
```

Note that the files in the module `circulation` all live in the subdirectory `circulation`. Thus the file `package.lisp` in the module `circulation` is a different one than that a level higher.

A module can have as components both files and other modules, which in turn can have files and modules as components. It is important to note that dependencies can only be defined *inside a given set of components*. So, the file `tail.lisp` cannot depend on the file `assorted-solids`, which is a component of a submodule.

2.3 A system that depends on other systems

Such a system will look exactly like a regular one, except for an additional `:depends-on` option. For instance

```
(defsystem cow
  ;; ...
  :components (...)
  :depends-on ("other-system"))
```

3 How to use system definition files

System definition files live in the directory where the corresponding piece of software lives. However, you do not need to have that directory as your working directory to be able to build and load said software. You only need to put a *symbolic link* to the system definition file in a directory where ASDF searches. The list of directories where

The usual setup is as follows. To begin with, you need to have ASDF loaded. In some implementations this is just a matter of writing

```
(require 'asdf)
```

whereas in others you might need to install it yourself first. . You can get it, in a single file, from <http://www.cloiki.net/asdf>, and put it somewhere reasonable, for instance in the directory `/home/chicken/lisp/utils/`. You may also want to compile that file.

Now, somewhere in the init file of your Common Lisp implementation a variation of the following passage should appear.

```
(load "/home/chicken/lisp/utils/asdf")

(setf asdf:*central-registry*
  ;; Default directories, usually just the ‘‘current directory’’
  (*default-pathname-defaults*

  ;; Additional places where ASDF can find
  ;; system definition files
  #p"/home/chicken/lisp/systems/"))
```

```
#p"/usr/share/common-lisp/systems/"))
```

The command to build and load system cow is

```
(asdf:operate 'asdf:load-op 'cow)
```

If the file `cow.asd` happens to be in the current working directory, the build and load process will start there. If not, ASDF will search through the directories in the central registry, and look for a system definition file named `cow.asd`, or for a symbolic link to one. If it finds the latter, it will follow the link to the original file, and run the build process in the corresponding directory. If it finds the file, it will run the build process in the directory where it finds it.

So, if you make a symbolic link in `/home/chicken/lisp/systems/` to the `cow` system definition file, by executing (for example)

```
$ cd <where-your-system-defs-are>
$ ln -s /home/chicken/code/cow/cow.asd
```

Then you can build and load the `cow` software without having to be in the directory where this software lives simply by issuing the command

```
(asdf:operate 'asdf:load-op 'cow)
```

References

- [1] D. Barlow et al., “asdf Manual”, <http://constantly.at/lisp/asdf/>